**Description**

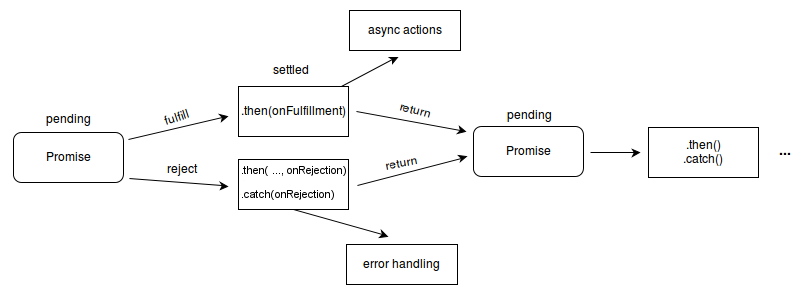
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

A pending promise can either be *fulfilled* with a value or *rejected* with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

As the [Promise.prototype.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) and [Promise.prototype.catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) methods return promises, they can be chained.



**Not to be confused with:** Several other languages have mechanisms for lazy evaluation and deferring a computation, which they also call "promises", e.g. Scheme. Promises in JavaScript represent processes that are already happening, which can be chained with callback functions. If you are looking to lazily evaluate an expression, consider the [arrow function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) with no arguments: f = () => *expression* to create the lazily-evaluated expression, and f() to evaluate.

**Note**: A promise is said to be *settled* if it is either fulfilled or rejected, but not pending. You will also hear the term *resolved* used with promises — this means that the promise is settled or “locked-in” to match the state of another promise. [States and fates](https://github.com/domenic/promises-unwrapping/blob/master/docs/states-and-fates.md) contain more details about promise terminology.

Chained Promises

The methods promise.then(), promise.catch(), and promise.finally() are used to associate further action with a promise that becomes settled. These methods also return a newly generated promise object, which can optionally be used for chaining; for example, like this:

const myPromise =

(new Promise(myExecutorFunc))

.then(handleFulfilledA,handleRejectedA)

.then(handleFulfilledB,handleRejectedB)

.then(handleFulfilledC,handleRejectedC);

// or, perhaps better ...

const myPromise =

(new Promise(myExecutorFunc))

.then(handleFulfilledA)

.then(handleFulfilledB)

.then(handleFulfilledC)

.catch(handleRejectedAny);

Handling a rejected promise too early has consequences further down the promise chain. Sometimes there is no choice because an error must be handled immediately; in such cases we must throw something, even if it is a dummy error message like throw -999, to maintain error state down the chain. On the other hand, in the absence of an immediate need it is simpler to leave out error handling until a final .catch() statement.

The termination condition of a promise determines the "settled" state of the next promise in the chain. Any termination other than a throw creates a "resolved" state while terminating with a throw creates a "rejected" state.

handleFulfilled(value) { /\*...\*/; return nextValue; }

handleRejection(reason) { /\*...\*/; throw nextReason; }

handleRejection(reason) { /\*...\*/; return nextValue; }

The returned nextValue can be another promise object, in which case the promise gets dynamically inserted into the chain.

When a .then() lacks the appropriate function that returns a Promise object, processing simply continues to the next link of the chain. Therefore, a chain can safely omit every handleRejection until the final .catch(). Similarly, .catch() is really just a .then() without a slot for handleFulfilled.

The promises of a chain are nested like Russian dolls, but get popped like the top of a stack. The first promise in the chain is most deeply nested and is the first to pop.

(promise D, (promise C, (promise B, (promise A) ) ) )

When a nextValue is a promise, the effect is a dynamic replacement. The return causes a promise to be popped, but the nextValue promise is pushed into its place. For the nesting shown above, suppose the .then() associated with "promise B" returns a nextValue of "promise X". The resulting nesting would look like this:

(promise D, (promise C, (promise X) ) )

A promise can participate in more than one nesting. For the following code, the transition of promiseA into a "settled" state will cause both instances of .then() to be invoked.

const promiseA = new Promise(myExecutorFunc);

const promiseB = promiseA.then(handleFulfilled1, handleRejected1);

const promiseC = promiseA.then(handleFulfilled2, handleRejected2);

An action can be assigned to an already "settled" promise. In that case, the action (if appropriate) will be performed at the first asynchronous opportunity. Note that promises are guaranteed to be asynchronous. Therefore, an action for an already "settled" promise will occur only after the stack has cleared and a clock-tick has passed. The effect is much like that of setTimeout(action,10).

const promiseA = new Promise( (resolutionFunc,rejectionFunc) => {

resolutionFunc(777);

});

// At this point, "promiseA" is already settled.

promiseA.then( (val) => console.log("asynchronous logging has val:",val) );

console.log("immediate logging");

// produces output in this order:

// immediate logging

// asynchronous logging has val: 777

**Constructor**

[**Promise()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise)

Creates a new Promise object. The constructor is primarily used to wrap functions that do not already support promises.

**Static methods**

[**Promise.all(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

Wait for all promises to be resolved, or for any to be rejected.

If the returned promise resolves, it is resolved with an aggregating array of the values from the resolved promises, in the same order as defined in the iterable of multiple promises.

If it rejects, it is rejected with the reason from the first promise in the iterable that was rejected.

[**Promise.allSettled(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled)

Wait until all promises have settled (each may resolve or reject).

Returns a Promise that resolves after all of the given promises have either resolved or rejected, with an array of objects that each describe the outcome of each promise.

[**Promise.any(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any)

Takes an iterable of Promise objects and, as soon as one of the promises in the iterable fulfills, returns a single promise that resolves with the value from that promise.

[**Promise.race(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)

Wait until any of the promises is resolved or rejected.

If the returned promise resolves, it is resolved with the value of the first promise in the iterable that resolved.

If it rejects, it is rejected with the reason from the first promise that was rejected.

[**Promise.reject(reason)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)

Returns a new Promise object that is rejected with the given reason.

[**Promise.resolve(value)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)

Returns a new Promise object that is resolved with the given value. If the value is a thenable (i.e. has a then method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise, the returned promise will be fulfilled with the value.

Generally, if you don't know if a value is a promise or not, [Promise.resolve(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) it instead and work with the return value as a promise.

**Instance methods**

[**Promise.prototype.catch()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch)

Appends a rejection handler callback to the promise, and returns a new promise resolving to the return value of the callback if it is called, or to its original fulfillment value if the promise is instead fulfilled.

[**Promise.prototype.then()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)

Appends fulfillment and rejection handlers to the promise, and returns a new promise resolving to the return value of the called handler, or to its original settled value if the promise was not handled (i.e. if the relevant handler onFulfilled or onRejected is not a function).

[**Promise.prototype.finally()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/finally)

Appends a handler to the promise, and returns a new promise that is resolved when the original promise is resolved. The handler is called when the promise is settled, whether fulfilled or rejected.

**Examples**

Basic Example

let myFirstPromise = new Promise((resolve, reject) => {

// We call resolve(...) when what we were doing asynchronously was successful, and reject(...) when it failed.

// In this example, we use setTimeout(...) to simulate async code.

// In reality, you will probably be using something like XHR or an HTML5 API.

setTimeout( function() {

resolve("Success!") // Yay! Everything went well!

}, 250)

})

myFirstPromise.then((successMessage) => {

// successMessage is whatever we passed in the resolve(...) function above.

// It doesn't have to be a string, but if it is only a succeed message, it probably will be.

console.log("Yay! " + successMessage)

});

Example with diverse situations

This example shows diverse techniques for using Promise capabilities and diverse situations that can occur. To understand this, start by scrolling to the bottom of the code block, and examine the promise chain. Upon provision of an initial promise, a chain of promises can follow. The chain is composed of .then() calls, and typically (but not necessarily) has a single .catch() at the end, optionally followed by .finally(). In this example, the promise chain is initiated by a custom-written new Promise() construct; but in actual practice, promise chains more typically start with an API function (written by someone else) that returns a promise.

The example function tetheredGetNumber() shows that a promise generator will utilize reject() while setting up an asynchronous call, or within the call-back, or both. The function promiseGetWord() illustrates how an API function might generate and return a promise in a self-contained manner.

Note that the function troubleWithGetNumber() ends with a throw(). That is forced because an ES6 promise chain goes through all the .then() promises, even after an error, and without the "throw()", the error would seem "fixed". This is a hassle, and for this reason, it is common to omit rejectionFunc throughout the chain of .then() promises, and just have a single rejectionFunc in the final catch(). The alternative is to throw a special value (in this case "-999", but a custom Error type would be more appropriate).

This code can be run under NodeJS. Comprehension is enhanced by seeing the errors actually occur. To force more errors, change the threshold values.

"use strict";

// To experiment with error handling, "threshold" values cause errors randomly

const THRESHOLD\_A = 8; // can use zero 0 to guarantee error

function tetheredGetNumber(resolve, reject) {

try {

setTimeout(

function() {

const randomInt = Date.now();

const value = randomInt % 10;

try {

if(value >= THRESHOLD\_A) {

throw new Error(`Too large: ${value}`);

}

} catch(msg) {

reject(`Error in callback ${msg}`);

}

resolve(value);

return;

}, 500);

// To experiment with error at set-up, uncomment the following 'throw'.

// throw new Error("Bad setup");

} catch(err) {

reject(`Error during setup: ${err}`);

}

return;

}

function determineParity(value) {

const isOdd = value % 2 ? true : false ;

const parityInfo = { theNumber: value, isOdd: isOdd };

return parityInfo;

}

function troubleWithGetNumber(reason) {

console.error(`Trouble getting number: ${reason}`);

throw -999; // must "throw" something, to maintain error state down the chain

}

function promiseGetWord(parityInfo) {

// The "tetheredGetWord()" function gets "parityInfo" as closure variable.

var tetheredGetWord = function(resolve,reject) {

const theNumber = parityInfo.theNumber;

const threshold\_B = THRESHOLD\_A - 1;

if(theNumber >= threshold\_B) {

reject(`Still too large: ${theNumber}`);

} else {

parityInfo.wordEvenOdd = parityInfo.isOdd ? 'odd' : 'even';

resolve(parityInfo);

}

return;

}

return new Promise(tetheredGetWord);

}

(new Promise(tetheredGetNumber))

.then(determineParity,troubleWithGetNumber)

.then(promiseGetWord)

.then((info) => {

console.log("Got: ",info.theNumber," , ", info.wordEvenOdd);

return info;

})

.catch((reason) => {

if(reason === -999) {

console.error("Had previously handled error");

}

else {

console.error(`Trouble with promiseGetWord(): ${reason}`);

}

})

.finally((info) => console.log("All done"));